第4章　进程间通信机制

内核用于协调进程间相互通信的活动。 Linux支持一部分进程间的通信 (Inter-Process Communication,IPC)机制。信号和管道是两种 IPC机制，但Linux也支持UNIX™ system V的IPC机制

4.1　信号机制

信号机制是UNIX系统使用最早的进程间通信机制之一，主要用于向一个或多个进程发异步事件信号，信号可以通过键盘中断触发、也可以由进程访问虚拟内存中不存在的地址这样的错误来产生。信号机制还可以用于 shell向它们的子进程发送作业控制命令。

系统内有一组可以由内核或其他的进程触发的预定义信号，并且这些信号都有相应的优先级。你可以使用 kill命令(kill-1)列出系统支持的所有信号。在作者的 Intel硬件平台的Linux系统上会产生如下的结果：

```
 1) SIGHUP       2) SIGINT     3) SIGQUIT     4) SIGILL
 5) SIGTRAP      6) SIGIOT     7) SIGBUS      8) SIGFPE
 9) SIGKILL     10) SIGUSR1   11) SIGSEGV    12) SIGUSR2
13) SIGPIPE     14) SIGALRM   15) SIGTERM    17) SIGCHLD
18) SIGCONT     19) SIGSTOP   20) SIGTSTP    21) SIGTTIN
22) SIGTTOU     23) SIGURG    24) SIGXCPU    25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF   28) SIGWINCH   29) SIGIO
30) SIGPWR
```

Alpha AXP硬件平台的Linux系统支持的信号数量与前面的不同。进程可以选择忽略上面的大多数信号，但 SIGSTOP和SIGKILL是不可忽略的。其中 SIGSTOP信号，使进程停止执行；而SIGKILL信号使进程中止。对于其他情况，进程可以自主决定如何处理各种信号：它可以阻塞信号；如果不阻塞，也可以选择由进程自己处理信号或者由内核来处理。由内核来处理信号时，内核对每个信号使用相应的缺省处理动作，例如：当进程收到 SIGFPE信号(浮点异常)时，内核的缺省动作是进行内核转贮 (core dump)，然后中止该进程。信号之间不存在内在的相对优先级。如果对同一个进程同时产生两个信号的话，它们会按照任意顺序提交给该进程，并且对同种信号无法区分信号的数量。例如：进程无法区别它收到了 1个还是42个SIGCONT信号。

Linux使用存贮在每个进程 task_struct结构中的信息实现信号机制， 它支持的信号数受限于处理器的字长，具有 32位字长的处理器有 32种信号，而像 Alpha AXP处理器有64位字长，最多可以有64种信号。当前未处理的信号记录在 signal域中，并把阻塞信号掩码对应位设置为阻塞状态。但对 SIGSTOP和SIGKILL信号来说，所有的信号都被设置为阻塞状态。如果一个被阻塞的信号产生了，就将一直保持未处理状态，直到阻塞被取消。 Linux中还包括每个进程如何处理每种可能信号的信息，这些信息被记录在 sigaction数据结构的矩阵中，由每个进程的task_struct指向sigaction矩阵。这些信息中包括处理信号例程的地址或者通知 Linux该进程选择

忽略信号还是由内核处理信号的标志。进程通过系统调用改变缺省信号的处理过程，这些系统调用会改变对应信号的sigaction结构和阻塞掩码。

并不是系统中的每个进程都可以向其他的进程发消息，只有内核和超级用户可以做到这一点。普通的进程只能向同一进程组或具有相同的 uid和gid的进程发送信号。信号可以通过设置task_struct结构signal域中相应中的位来产生。如果一个进程没有阻塞信号，正处于可中断的等待信号状态中，当等待的信号出现时 ,系统可以通过把该进程的状态变成运行状态 ,然后放入候选运行队列中的方法来唤醒它。通过上面的方法在下次调度时 ,进程调度器会把该进程作为候选运行进程进行调度。如果需要缺省处理的话， Linux可以优化信号的处理，例如：当出现SIGWINCH(X window焦点改变信号)信号时，如果没有进程的信号处理例程可以调用的话，系统会使用缺省处理过程。

信号产生后，并不立即提交给进程 ,它必须要等到进程再次被调度运行时。每当进程从系统调用中返回时，系统都会检查进程的 signal域和blocked域，以确定是否出现某些未阻塞的信号。这看起来非常不可靠，但实际上系统的每个进程都在不断地做系统调用，如向终端写字符。进程可以选择挂起在可中断的状态上，等待某一个它希望的信号出现， Linux的信号处理程序为当前每个未阻塞的信号查找 sigaction结构。

如果一个信号被设置为按缺省动作处理，那么内核会处理它。 SIGSTOP信号的缺省处理是把当前进程的状态改为停止状态，然后运行进程调度器选择一个新进程运行。 SIGFPE信号的缺省处理动作是对该进程进行内核转贮，然后中止该进程。相反，进程也可以指定自己的信号处理例程。在 sigaction结构中存贮有这个信号处理例程的地址，当信号产生时，这个例程就会被调用。内核必须调用信号处理例程，但如何调用是与处理器相关的。在调用信号处理例程时，所有的CPU必须要考虑到下面的几个问题：当前进程正在核态中运行，准备返回到用户态，而对信号处理例程的调用是由内核或系统例程来完成的。这个问题可以通过对栈和进程寄存器进行操作来解决，系统把进程的程序计数器置为进程信号处理例程的地址，并把例程的参数加到函数调用桢中或通过寄存器来传递参数。当进程重新开始运行时，信号处理例程就像被正常调用了一样。

Linux兼容POSIX标准，进程在某个信号处理例程被调用时，能指出哪些信号可以被阻塞。这意味着在调用进程信号处理例程时需要改变阻塞掩码，当信号处理例程结束时，阻塞掩码必须要恢复到初始值。因此 Linux增加了一次对清理例程的调用。清理例程按照信号处理例程的调用栈来恢复初始的阻塞掩码。在几个信号处理例程都需要被调用时， Linux也提供了优化方案。Linux把这些例程压入栈中，这样每当一个处理例程退出时，下一个处理例程立即被调用,当所有处理例程都完成后清理例程被调用。

4.2 管道

Linux shell通常支持重定向操作。例如对命令：

```
$ ls | pr | lpr
```

管道操作把列出目录中所有文件 ls命令的标准输出重定向为分页命令 pr的标准输入,接着pr命令的标准输出又被管道操作重定向为 lpr命令的标准输入 (lpr命令的作用是在缺省的打印机上打印)。管道是单向的字节流，它可以把一个进程的标准输出与另一个进程的标准输入连接起来。Linux的shell负责建立进程间的这些临时性的管道，而进程根本不知道这些重定向操作，

仍然按照通常的方式进行操作。

在Linux系统中，管道用两个指向同一个临时性 VFS索引节点的文件数据结构来实现。这个临时性的VFS索引节点指向内存中的一个物理页面。图 1-4-1表明每个文件数据结构包含指向不同文件操作例程向量的指针。一个例程用于写管道，另一个用于从管道中读数据。从一般读写普通文件的系统调用的角度来看，这种实现方法隐藏了下层的差异。当写进程执行写管道操作时，数据被复制到共享的数据页面中 ;而读进程读管道时，数据又从共享数据页中复制出来。Linux必须同步对管道的访问，使读进程和写进程步调一致。为了实现同步， Linux使用锁、等待队列和信号量这三种方式。

写进程使用标准的写库函数来写管道。使用文件操作库函数要求传递文件描述符来索引进程的文件数据结构集合。每个文件数据结构代表一个打开的文件或是一个打开的管道。Linux写系统调用使用代表该管道的文件数据结构指向的写例程，而写例程又使用代表该管道的VFS索引节点中保存的信息来管理写请求。

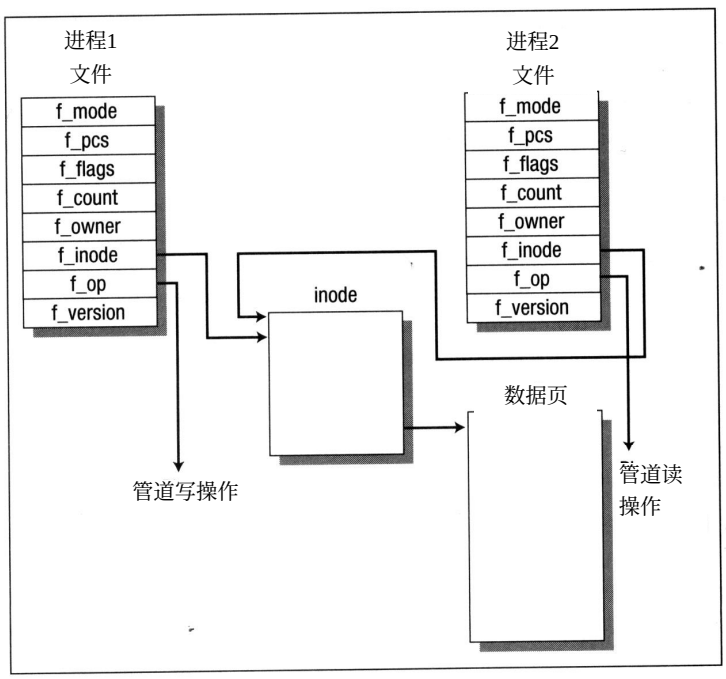

图1-4-1 管道

如果有足够大的空间把所有的数据写入管道中，并且该管道没有被读进程锁定，那么Linux为写进程锁定管道，把待写的数据从进程空间复制到共享数据页中。如果管道被读进程锁定或者没有足够大的空间存放数据，那么当前的进程被强制进入睡眠状态，放在管道对应的索引节点的等待队列中，然后系统调用进程调度器来选择合适进程进入运行状态。睡眠的进程是可中断的,它可以接收信号；也可以在管道中有足够大空间来容纳写数据或在管道被解锁时，被读进程唤醒。写数据完成后，管道的 VFS索引节点被解锁。系统会唤醒所有睡眠在读索引节点等待队列中的读进程。

从管道中读数据的过程与向管道中写数据非常相似。进程可以做非阻塞的读操作，但它依赖于打开管道的模式。进程使用非阻塞读时，如果管道中无数据或者该管道被锁定，读系

统调用会立即返回出错信息。通过这种办法，进程可以继续运行。另一种处理是进程在索引节点的等待队列中等待写进程完成。一旦所有的进程都完成了管道操作，管道的索引节点和共享数据页会立即被释放。

Linux也支持命名管道(named pipes)。因为这种管道遵循先进先出的规则，所以它也被称为FIFO(先进先出)管道。普通的管道是临时性的对象，而FIFO管道是通过mkfifo命令创建的文件系统中的实体。只要有适当的权限，进程就可以自由地使用FIFO管道。但FIFO管道的打开方式与普通管道有所不同：普通管道(包括两个文件数据结构：对应的VFS索引节点以及共享数据页)在进程每次运行时都会创建一次，而FIFO是一直存在的，需要用户打开和关闭。Linux必须处理读进程先于写进程打开管道、读进程在写进程写入数据之前读入这两种情况。除此之外，FIFO管道的使用方式与普通管道完全相同，都使作相同的数据结构和操作。

4.3 套接字

4.3.1 System V的进程间通信机制

Linux支持最早在UNIX System V中出现的三种进程间通信机制。它们是消息队列、信息量和共享存储器。这些System V的进程间通信机制使用相同的认证方法，即通过系统调用向内核传递这些资源的全局唯一标识来访问它们，Linux使用访问许可的方式核对对System V IPC对象的访问，这种方式与文件访问权限的检查十分相似。

System V IPC对象的访问权限是由该对象的创建者通过系统调用来实现的。Linux的每种IPC机制都把IPC对象的访问标识作为对系统资源表的索引，但访问标识不是一种直接的索引，而是由索引标识通过某些运算来产生的对象索引。

Linux系统中所有代表System V IPC对象的数据结构中都包括ipc_perm数据结构，在ipc_perm结构中有拥有者和创建者进程的用户标识和组标识、该对象的访问模式以及IPC对象的密钥。密钥的用处是确定System V IPC对象的索引标识。Linux系统中支持两种密钥：公共密钥和私有密钥。如果IPC对象的密钥是公共的，那么系统中的进程在通过权限检查后就可以得到System V IPC对象的索引标识。但要注意System V IPC对象不是通过密钥而是通过它们的索引标识来访问的。

4.3.2 消息队列

消息队列允许一个或多个进程向队列中写入消息，然后由一个或多个读进程读出(见图1-4-2)。Linux系统维护一个消息队列的表。该表是msgque结构的数组，数组中每个元素指向一个能完全描述消息队列的msqid_ds数据结构。一旦一个新的消息队列被创建，则在系统内存中会为一个新的msqid_ds数据结构分配空间，并把它插入到数组中。

每个msqid_ds结构都包含ipc_perm数据结构以及指向进入该队列的消息的指针。除此之外，Linux还记录像队列最后被更改的时间等队列时间更改信息。msqid_ds结构还包括两个等待队列；一个用于存放写进程的消息，另一个用于消息队列。

每次进程要向写队列写入消息时，系统都要把它的有效用户标识和组标识与该队列的

ipc_perm数据结构中的访问模式进行比较。如果进程可以写队列。那么消息会从进程的地址空间复制到一个 msg数据结构中，然后系统把该 msg数据结构放在消息队列的尾部。由于Linux限制写消息的数量和消息的长度，所以可能会出现没有足够的空间来存放消息的情况。这时当前进程会被放入对应消息的写等待队列中，系统调用进程调度器选择合适的进程运行。在该消息队列中有一个或多个消息被读出时，睡眠的进程会被唤醒。

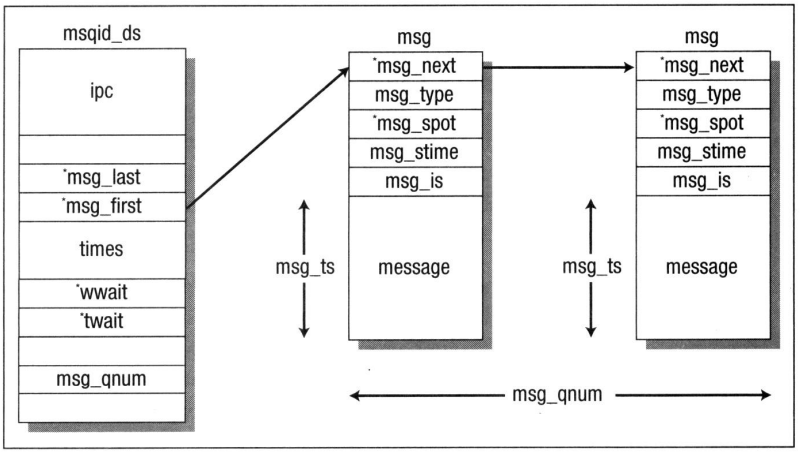

图1-4-2 System V IPC 消息队列

从队列中读消息的过程与前面相似，进程对写队列的访问权限会再次被核对。一个读进程可以选择获得队列中的第一个消息而不考虑消息的类型，还是读取某种特别类型的消息。如果没有符合要求的消息的话，读进程会被加入到该消息的读等待队列中，系统唤醒进程调度器调度新进程运行。一旦有新消息被写入消息队列。睡眠的进程被唤醒，并再次运行。

4.3.3 信号量

最简单的信号量是内存中的一个区域，它的值可以被多个进程执行 test_and_set操作(一种具有原子性的系统调用，用于测试某一地址的值然后再更改它)。test_and_set操作对每个进程来说是不可中断的，即具有原子性的操作。一旦一个进程执行该操作，其他的任何进程都不能打断它。Test_and_set操作的结果是对当前信号量的值进行增量操作，但增量可以是正的，也可以是负的。根据test_and_set操作的结果，进程可能会进入睡眠状态，等待其他进程改变信号量的值。信号量能用于实现关键段操作 (关键段指一段关键的代码段，同一时间内只有一个进程能执行该段操作)。

假如有许多相互协作的进程从一个数据文件中读取或写入记录，你会要求对文件的访问应是严格相互同步的。这样可以在文件操作的代码外面，使用两个信号量操作，并把信号量的初始值置为1。第一个操作是测试并减少信号量的值；第二个操作是测试并增加信号量的值。当第一个进程访问文件时，它会减少信号量的值，使信号量的值变为 0，这样第一个进程可以成功的进行文件操作了。这时若有另一个进程要访问文件而去减小信号量的值，信号量的值变为－1，从而这个进程被挂起，等待第一个进程完成数据文件的操作。当第一个进程完成文件操作时，它会增加信号量的值，使其再次变为 1。现在系统会唤醒所有的等待进程，这时第二个要访问文件进程的减1操作会成功。

System V的每个IPC信号量对象都对应一个信号量数组，在 Linux中用semid_ds数据结构来表示它(见图1-4-3)。系统中所有的 semid_ds数据结构都被一个叫 semary的指针向量指向。在每个信号量数组中都有 sem_nsems域，这个域由sem_base指向的sem数据结构来描述。所有允许对System V IPC信号量对象的信号量数组进行操作的进程，都必须通过系统调用来执行这些操作。在系统调用中可以指出有多少个操作。而每个操作包含三个输入项：信号量的索引、操作值和一组标志位。信号量索引是对信号量数组的索引值，而操作值是加到当前信号量值上的数值。首先 Linux会测试是否所有的操作都会成功 (操作成功指操作值加上信号量当前值的结果大于0，或者操作值和信号量的当前值都是 0)。如果信号量操作中有任何一个操作失败，Linux在操作标志没有指明系统调用为非阻塞状态时，会挂起当前进程。如果进程被挂起了，系统会保存要执行的信号量操作的状态，并把当前进程放入等待队列中。 Linux通过在栈中建立一个sem_queue数据结构，并填入相应的信息的方法来实现前面的保存信号量操作状态的。新的 sem_queue数据结构被放在对应信号量对象的等待队列的末尾 (通过使用sem_pendinging和sem_pending_last指针)，当前进程被放在sem_queue数据结构的等待队列中，然后系统唤醒进程调度器选择其他进程执行。

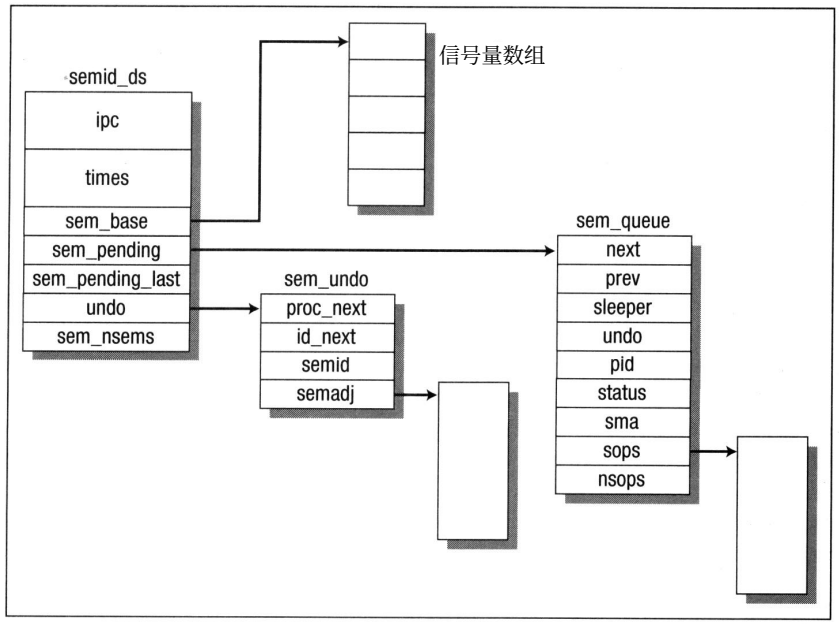

图1-4-3 System V IPC 信号量

如果所有的信号量操作都成功了，那么当前进程就不必挂起了。 Linux会继续运行当前进程，对信号量数组中的对应成员执行相应的操作。接着 Linux会查看那些处于等待状态被挂起的进程，以确定它们是否能继续持行信号量操作。 Linux会逐个查看等待队列中的每个成员，测试它们现在能否成功地执行信号量操作。如果有进程可以成功地执行了 ，Linux会删除未完成操作列表中对应的sem_queue数据结构，对信号量数组执行信号量操作，然后唤醒睡眠进程，将其放入就绪队列中。 Linux不断地查找等待队列，直到没有可成功执行的信号量操作并且也没有可唤醒的进程为止。

但信号量存在着死锁(deadlock)的问题，当一个进程进入了关键段，改变了信号量的值后，

由于进程崩溃或被中止等原因而无法离开关键段时，就会造成死锁。 Linux通过为信号量数组维护一个调整项列表来防止死锁。主要的想法是在使用调整项后，信号量会被恢复到一个进程的信号量操作集合执行前的状态。调整项被保存在 sem_undo数据结构中，而这些sem_undo数据结构则按照队列的形式放在 semid_ds数据结构和进程使用信号量数组的 task_struct数据结构中。

每一个单独的信号量操作都要求建立相应的调整项。 Linux为每个进程的每个信号量数组至多维护一个sem_undo数据结构。如果还没有为请求的进程建立调整项，那么当需要时，系统会为它创建一个新的 sem_undo数据结构。 sem_undo数据结构被加入到该进程的 task_struct数据结构和信号量数组的 semid_ds数据结构的队列中。一旦对信号量数组中某些信号量执行了相应的操作，那么该操作数的负值会被加入到该进程 sem_undo结构调整项数组的与该信号量对应的记录项中。因此，如果操作值是 2的话，那么－2就被加到该信号量的调整项中。

当进程被删除时，退出时 Linux会用这些sem_undo数据结构集合对信号量数组进行调整。如果信号量集合被删除了，那么这些 sem_undo数据结构还存在于进程的task_struct结构的队列中，而仅把信号量数组标识标记为无效。在这种情况下，信号量清理程序仅仅丢掉这些数据结构而不释放它们所占用的空间。

4.3.4 共享存储区

共享存储区允许一个或多个进程通过在它们的虚地址空间中同时出现的存储区进行通信。虚地址空间的页是通过共享进程的页表中的页表项来访问的。共享存储区不需要在所有进程的虚存中占有相同的虚地址。像所有的 System V IPC对象一样，共享存储区的访问控制是通过密钥和访问权限检查来实现的。一旦某一内存区域被共享了，系统就无法检查进程如何使用这部分内存区域。因此系统必须使用 System V信号量等其他的机制来同步对存储器的访问。

每个新创建的共享存储区由shmid_ds数据结构来表示，并被记录在shm_segs向量中(见图1-4-4)。shmid_ds数据结构中包含共享存储区的大小、当前使用该共享存储区的进程数目以及共享存储区如何映射到进程地址空间等信息。共享存储区的创建者设置对该共享存储区的访问许可权限，并确定它的密钥是公用的还是私有的。如果一个进程有足够的访问权限，就可以将共享存储区锁定到物理存储区域上。

每个想访问共享存储区的进程必须先通过系统调用，将该共享存储区连接到它的虚地址空间中。这个操作会创建一个描述该进程共享存储区的 vm_area_struct数据结构。进程既可以指定共享存储区放在它的虚地址空间的位置，也可以由 Linux自动选择一个足够大的自由空间。新的vm_area_struct数据结构被放入由shmid_ds指向的vm_area_struct结构的双向链表中。这个双向链表由vm_area_struct结构中的vm_next_shared指针和vm_prev_shared指针链接在一起。在执行连接操作时，系统实际上还没有创建该共享存储区，只有在第一个进程要访问共享存储区时，系统才会执行实际的创建工作。

当某一进程第一次访问共享存储区的某一页时，系统会产生一个页失效。 Linux在处理页失效时，它会找到描述该页的vm_area_struct数据结构。在vm_area_struct结构中包含处理这种共享存储区页失效的例程的句柄。共享存储区页失效处理例程会为 shmid_ds结构查找页表项的列表，以确定共享存储区中的这个页是否存在。如果不存在， Linux分配一个物理页，并为该页创建页表项。这个新的页表项会被同时保存到当前进程的页表和 shmid_ds结构中。这种

处理方法使得在下一个进程访问这个页，产生页失效时，共享存储区页失效处理例程会再次使用被分配的物理页。因此，第一个访问共享存储区某个页的进程会导致系统创建该共享页，而其他访问该共享页的进程仅仅会把该页增加到它们的虚地址空间中。

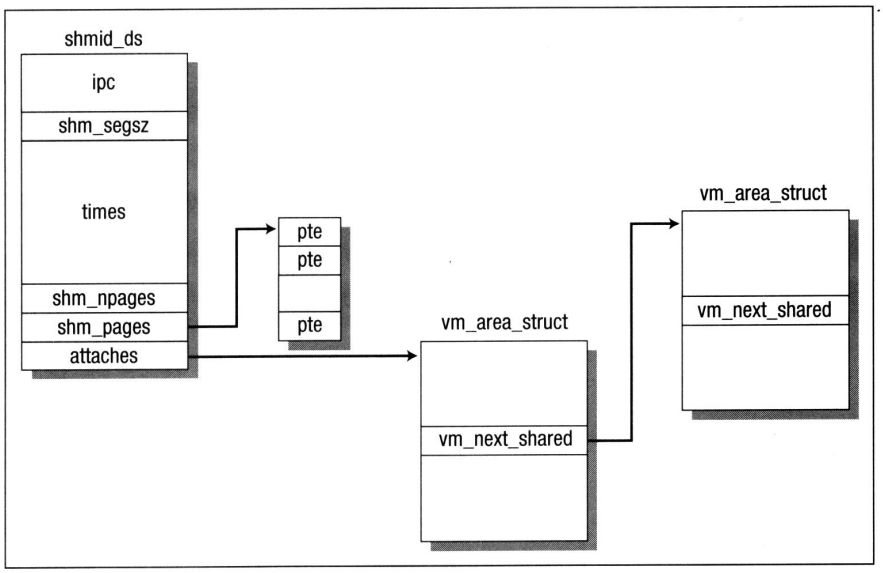

图1-4-4 System V IPC 共享存储区

当进程不再使用共享存储区时，进程会执行分离操作。只要还有其他的进程仍在使用这块存储区，分离操作就只会影响当前进程。进程的 vm_area_struct结构会被从shmid_ds结构中删除、释放掉，系统更新当前进程的页表以使原来被共享的虚地址区域无效。在最后一个使用共享存储区的进程执行分离操作时，处在物理存储器中的共享页面才会被释放掉，同时该共享存储区对应的shmid_ds数据结构也会被释放。

当共享存储区没有被锁定在物理存储区上时，会产生更复杂的情况。这时如果内存利用率比较高，共享存储区的页面会被交换到系统的磁盘交换区中。如何将共享存储区交换进和交换出物理存储器已在第2章中讨论过了，详细情况请参见第 2章。